Intel® Renewable Authentication Agent System 1.2

Developer's Guide

Please direct any comments or corrections to your support contact at Intel.
Any information herein is subject to change without notice.

Document Last Modified 4.12.99

This Guide is provided "as is" with no warranties, express or implied, including but not limited to any implied warranty of merchantability, fitness for a particular purpose, or freedom from infringement.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the described subject matter. The furnishing of this document does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Intel Corporation assumes no responsibility for errors or omissions in this document; nor does Intel make any commitment to update the information contained herein. Distribution of this document is limited to that permitted by your Nondisclosure Agreement with Intel.

To protect the secrets during delivery of the CD to customer sites, the secrets.txt file is encrypted with PGP and GNUPG (GNU Privacy Guard) using the CAST5 algorithm and must be decrypted before use. Encryption laws vary from country to country. Please check your local laws before purchasing or downloading encryption software.

“Year 2000 Capable.” See Support at www.intel.com.

Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497

Contents

Section 1: Using This Guide

Intended Audience

This document is intended for software developers and software architects interested in enhancing the security of an application with "machine-to-machine" authentication. This document describes our "Tamper Resistant Authentication Agent" technology and specifies the programming interfaces used to integrate this technology into existing applications.

Typographic Conventions

All information in this guide is divided into numerical sections and subsections, which are listed in the Table of Contents.

```
Courier font is used to delineate actual programming code.
```

NOTE: Throughout the guide, helpful facts are set apart from the rest of the text like this note.

Definitions of Commonly Used Terms

This Guide contains several terms that the reader may not be familiar with. Here are the definitions for common terms and acronyms used throughout this guide.

Agent: Instance of Tamper Resistant Software that can be used to securely verify the identity of a remote machine.

Agent Pool: A large set of unique TRS Authentication Agents used to authenticate remote Pentium® III processor-based machines. Each agent is unique, and is used for a short period of time (for example, 20 minutes) before it expires. Using a large number of unique agents minimizes the chance that an agent can be successfully attacked during its lifetime.

Agent Secret: A 512-byte random number embedded within the agent. It is used to generate a Message Authentication Code for the machine identifier that the agent retrieves.

ClientID: A number that is used as a unique identifier for a client machine for a particular server. The ClientID is based on the Pentium III processor serial number (PS#) but is a different number, thereby protecting the machine user's privacy.

Cryptographic Hash: A hash function in general operates on an arbitrary length message M, and generates a fixed length output S. Cryptographic hashes are hashes that exhibit these additional properties:

- One-way: Given M, it is easy to compute S=H(M) but impractical to compute M from S.
- Collision-Resistance: Given a message M, it is difficult to find another message M_x such that $H(M) = H(M_x)$.

Machine-to-Machine Authentication: The process by which an application verifies the identity of the remote machine with which it is communicating. In general, authentication is the first step in performing secure communication.

Message Authentication Code (MAC): A way to digitally sign a message M, using a cryptographic hash (SHA1) and a secret key (K).

Intel® Processor Serial Number (PS#): Silicon devices in Intel's Pentium III processors are embedded with a unique number during manufacture. The Intel processor serial number serves as an identifier for the processor, and, by association, its system.

Registration: The initial login of a client machine, in which its ClientID is stored in the server's database for future verification of the client.

Secret: See definition for "Agent Secret."

ServiceID: A variable length string assigned to a particular Web site or service that uses tamper-resistant agents.

SHA1: A U.S. government standard for computing a cryptograph hash that is used extensively in the components in this SDK. There are no known cryptographic attacks against SHA1 as of this publication.

Tamper Resistant Software (TRS): A term used to describe a class of software that has been designed and manufactured to do one or more of the following:

1) Make it difficult to observe the execution of the code.
2) Hide a secret key in software through both space (in the code) and time (frequent renewal).
3) Make it difficult to modify the intended behavior of a piece of software.

Token Based Authentication: The process of verifying the identity of a remote machine using a hardware security device.

TRS Authentication Agent: Instance of Tamper Resistant Software that can be used to securely verify the identity of a remote machine.

Verification: The process of confirming that a client machine is a known and trusted machine. This is accomplished by validating the machine's authentication code against information that was stored in the database during registration.

Section 2: Introduction

Overview of Security Concerns

More and more companies are using the Internet to communicate and conduct business, both within their organizations and externally with business partners, suppliers, customers and consumers. While the Internet enables new and more efficient methods of conducting business, it also exposes an organization to new security risks.

Authentication is the process by which authorized users are distinguished from unauthorized users before access to content and services is granted. Applications have traditionally relied on passwords for authentication; however, password-based authentication has several well-documented limitations. End-users often forget their passwords, choose simple passwords that are easy to guess, or share them with unauthorized users. These factors, along with high administration costs, limit the effectiveness of passwords for providing "secure" access control.

"Machine-to-Machine Authentication" provides a robust second factor of authentication that is difficult to spoof or share, and is relatively inexpensive to deploy and manage when compared to other forms of token-based authentication. The introduction of a processor serial number (PS#) in the Pentium III processors provides a useful building block for building such a machine-based authentication system.

However, the PS# feature by itself does not provide a complete security solution. For example, if a server asked software on the client machine for the system identifier, a skilled attacker could "patch" the software to respond with any system identity the attacker wished. To overcome this limitation, "Tamper-Resistance" techniques are used.

The Intel® Renewable Authentication Agent System provides the basic software components and tools required to integrate Tamper-Resistant Authentication agents into existing applications to perform secure Machine-to-Machine Authentication.

Sample Case Studies

The Renewable Authentication Agent System can be used in a variety of applications. A few possible scenarios are:

- **Secure Document Distribution:** A company wants to distribute sensitive documents to mobile sales forces over the Internet.

- **Secure Communications:** A company wants to broadcast training videos and live presentations over the Internet to distributed sites, ensuring that only authorized employees have access to the content.

- **Secure E-Commerce Transactions:** A company has installed an online inventory system and wants to ensure that only authorized personnel at their resellers can place orders or access sensitive pricing information.

- **Secure Chat:** A chat room administrator bans an abusive user, and needs to ensure that this user cannot sign up again with a new login name.

The common thread between these applications is that the systems must be built with the assumption that a potential attacker would have complete control of a remote system and the attacker is assumed to be actively attempting to break the authentication mechanisms to gain access to the content or services.

Agent Refresh

Intel recommends refreshing the agent being used by the server at a fixed interval. This ensures that an attacker would not have time to reverse-engineer an agent and attack the server. Each agent is constructed separately, so no two CDs contain any of the same agents, so the more often your used agents are regularly replaced by new ones, the higher your level of security.

SDK Contents

This CD contains the following files and directories:

Files

readme.txt
service.id
config.txt
secrets.txt.pgp
license.txt

Directories

NOTE: There is a readme.txt file in most directories that can give you more information about their structure and contents.

\agents
\bin
\sdk
  \agents
  \bin
  \docs
  \include
  \lib
  \linux-i386
  \src

For a description of the contents, see "Section 3: Development Guidelines."

What is Tamper Resistant Software?

Intel has developed a Tamper Resistant Software (TRS) technology. This technology helps create software modules that are resistant to observation and modification. It is designed to operate as intended even in the presence of a malicious attack. TRS design principles are based on the need to hide a secret in software and ensure that unauthorized recovery or alteration of the secret is difficult. The principles we use to build our tamper-resistant agents are:

- Disperse secrets in both time and space
- Conceal operations
- Actively recognize and disable well-known debugging mechanisms.
- Renew the modules regularly

None of these principles alone guarantees tamper resistant software. Tamper resistance is built from many applications of these ideas aggregated into a single software entity.

TRS Agents: Authentication Process

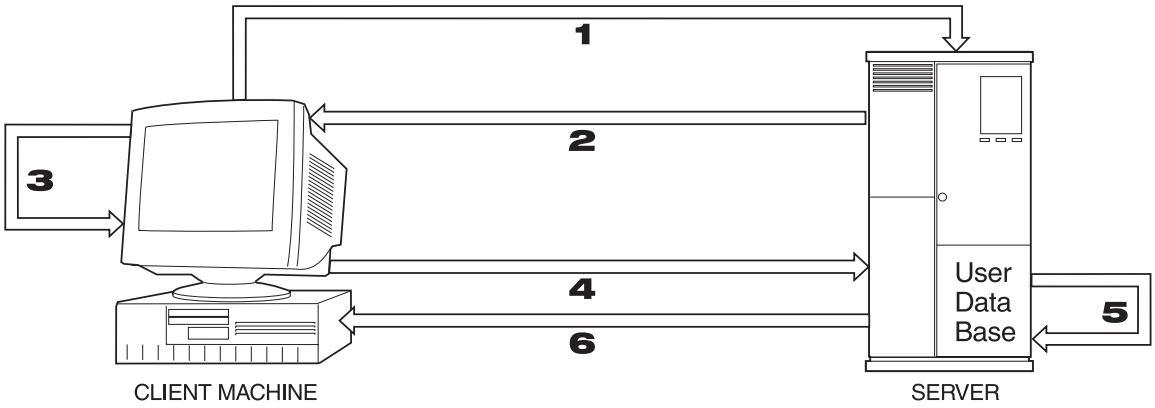


FIGURE 1: TRS AGENTS AUTHENTICATION PROCESS

1. A user accesses the system.
2. The server sends an Authentication Agent to the client.
3. The agent obtains the processor serial number and computes a hash.
4. That hash (which is a MAC) is sent back to the server.
5. The hash is verified against stored data on the server.
6. The server accepts or rejects access to the client machine, depending on whether the stored data matches the data returned by the agent.

Initial Registration Process

The first time a user logs into the system, the server sends an authentication agent to the client. The agent gets the processor serial number and computes a hash. This hash (which is a MAC) is sent back to the server, where it is stored in a database for future verification of that client machine.

Using Agents After Initial Registration

For users who have been registered, user authentication needs to be done for all subsequent login sessions. Authentication consists of two parts: secure data collection on the client and validation on the server.

Data Collection on the Client Side

The server maintains a pool of TRS agents (the contents of this CD), each of which uses a unique embedded secret key. Each time a user logs in, the server sends an agent to the client. This agent reads the processor serial number and computes a hash value of the processor_number and server_id, which it sends back to the server.

H2 = Hash(Hash(processor_number, server_id), session_id, secret, random_num)random_num

Since the session_id variable varies from one user session to the next, the same agent returns a different value for each session. To protect the client from providing a global ID, a 32-bit random number is generated on the client side. The result of the original hashing is a 160-bit (20-byte) value. The value (H2) returned to the server as a 24-byte value, composed of the 20-byte hash result plus a 4-byte random number.

Validation on the Server Side

The server retrieves the hashed value returned by the verification agent and stores it in a temporary location. It also retrieves the stored value (computed and saved during the user's initial registration) of the ClientID hash (processor_number,server_id) from the database. This value is matched with the value returned from the client. If the two values match, the client is authenticated.

Using TRS Agents to Verify Intel Processor Serial Numbers

The processor number feature is used to reliably identify a Pentium III processor in a client machine by reading its 96-bit processor number and storing a unique ClientID on the server for future verification. To protect the client's privacy, the raw processor number is not retrieved. Instead, the processor number is transformed on the client by means of a collision-free, one-way hash function. This hashed value gets stored on the server.

NOTE: Because the ClientID is a hash (service_id, PS#), it protects the client's privacy by ensuring that the ClientID cannot be used to track users between services. A cryptographically secure hash (SHA1) is used, and each service has a different service_id. Based on the cryptographic strength of SHA1, it is not possible to tell if two different ClientIDs were created from the same PS#.

Authenticating Multiple Users Of A Single Machine

Multiple users who are sharing one machine may have accounts at your service. Your authentication framework needs to take that into consideration, since more than one username will be associated with a single Client ID.

Authenticating A Single User of Multiple Machines

One user may connect from multiple machines. Your authentication framework needs to take that into consideration, since more than one Client ID will be associated with the single username.

Section 3: Development Guidelines

This section will discuss each of the components in the system architecture, and talk about the various issues involved in using the Tamper Resistant Agents in conjunction with these third-party components and your existing authentication framework..

System Architecture Overview

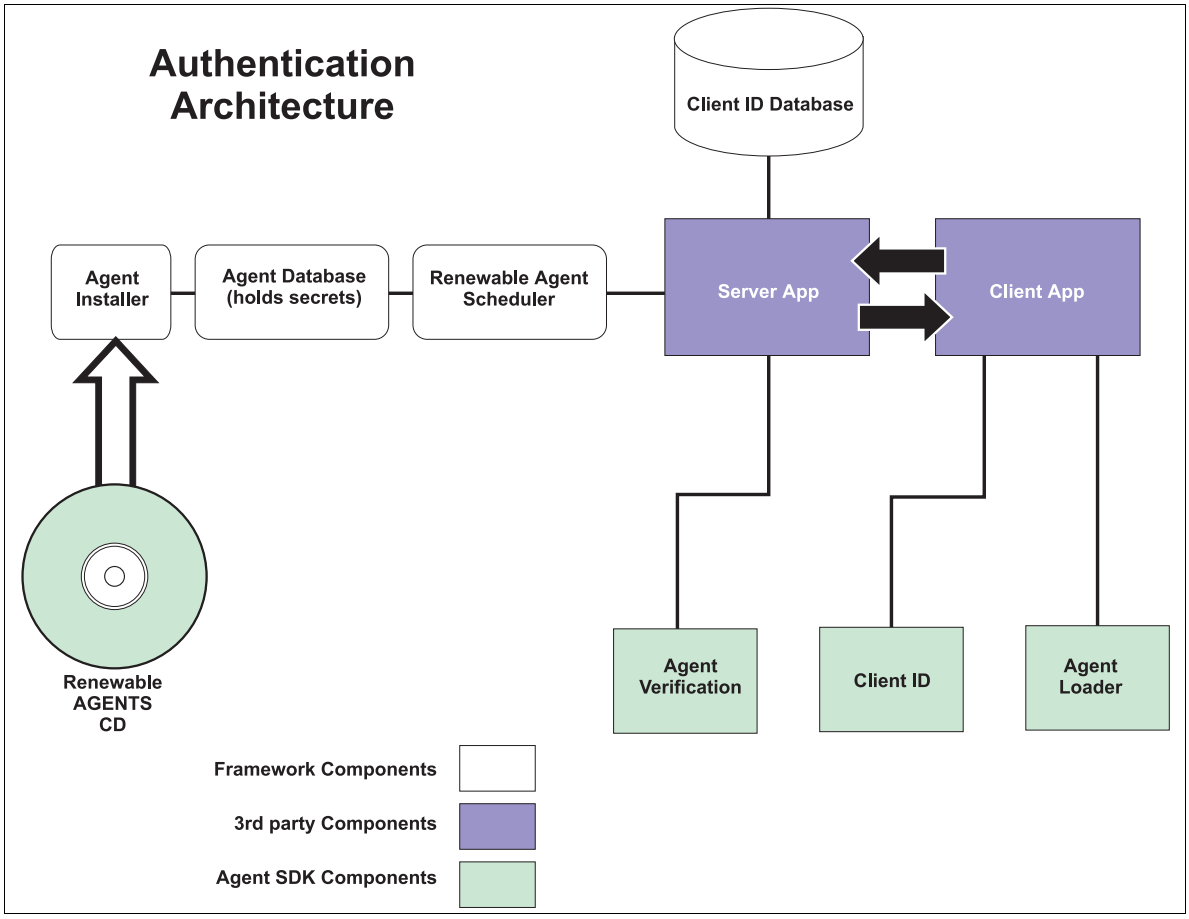
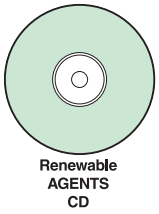


FIGURE 2: AUTHENTICATION ARCHITECTURE

In this figure, components in white are part of the framework, or glue layer, that connects the Intel SDK to an authentication system, and those in purple (or dark grey) are third-party components. Finally, components in green (or light grey) are part of the Renewable Authentication Agent System.

System Components



Renewable Agents CD

The CD contains these files and directories:

readme.txt
Summarizes the contents of the agent CD and outlines the revision history of the SDK.

service.id
Each agent contains a service identifier. This value is encoded on the CD in a service.id file. It contains a text string with no return or line feed characters. The service identifier provides a method of identifying the service.

The client application must provide this value to the GetClientID() function in clientid.dll. It is also used by the ValidateAuthentication()function in authverify.dll or authverify.so.

NOTES

- This ClientID value is used to register users. The model here is that each service registers users separately. Thus if several servers need to provide services to the same list of registered users, they must all use the same service identifier.
- The Customer Evaluation version of the CD does not have a unique service id, and should be considered insecure.

Once client machines are registered using a particular service_id, the service_id cannot be changed without re-registering the client machines. All future agent CDs sent to this site will use the same service_id.

config.txt
Config.txt is a machine-readable configuration file that lists several attributes of the agent CD. This file must be referenced by your Agent Installer tool to get information about the agents being used. For more information, see "Agent Installer."

secrets.txt.pgp
Every agent contains a unique secret. This secret is shared between the agent and the server through the secrets.txt file. It is this secret that enables validation of the agents' returned authentication code. To protect the secrets during delivery of the CD to customer sites, the secrets.txt file is encrypted for use with PGP (Pretty Good Privacy) and GNUPG (GNU Privacy Guard) using the CAST5 algorithm. These tools have the following advantages:

1. PGP is based on standard encryption techniques and has been widely scrutinized for its security. It is generally considered stronger than other ad-hoc encryption methods (e.g. pkzip).
2. PGP implementations are widely available at a minimal cost across the world. This ensures that our customers will be able to maintain the security of their sites while minimizing the cryptographic import/export issues.

NOTE: Encryption laws vary from country to country. Please check your local laws before purchasing or downloading encryption software.

The password used to decrypt the file will be delivered to an assigned administrator at your site separately from the CD. Please contact the appropriate Intel representative to get the password.

To get PGP, please see the following URLs:
www.nai.com/default_pgp.asp
dir.yahoo.com/Computers_and_Internet/Security_and_Encryption (click on PGP)

The secrets.txt file can also be decrypted using GNU Privacy Guard, which is available on their web site at:
www.gnupg.org

Once decrypted, the secrets.txt file contains the following:
1. Comment lines. These start with the # character and end at the next carriage return.
2. Blank lines. These may be empty, or have spaces or tabs.
3. Secret label. The word "SECRET" is followed by a space then a decimal number. For example, SECRET 4 matches the secret used by /agents/auth4.agent.
4. Secret values. This are provided in groups of 4 bytes, to be read from left to right. The value a49d8ff1 translates directly to the bytes 0xa4, 0x9d, 0x8f, 0xf1 in that order.

license.txt
This file contains the appropriate license for the CD.

Directories

\agents
Contains agents. The files are named auth*n*.agent, where *n* represents the number of that agent. For example, on a CD with 7200 agents, Auth0.agent is the first agent in the directory and auth7199.agent is the last agent in the directory.

Agents on the CD are provided in a "packaged" form and can be accessed using the trsagent.lib. One advantage of using them in this form is that they are smaller than the DLL versions. However, we have also provided a tool in the SDK (insert_agent.exe) that injects the agent into a template DLL (agent_shell.dll).

> **NOTE:** The number of agents is likely to change from release to release. The number of agents on the CD is in the config.txt file.

See "Section 4: Sample Application" for more information on accessing "packaged" versus DLL agents.

\bin
Contains copies of the client and server side components. Libraries are exactly the same binaries as the ones provided in the SDK portion of the CD. They are placed in this directory to assist automated installs. Please refer to the config.txt file for more information about the versions of these components.

\sdk
Contains the software development kit components.

\sdk\agents
Contains sample agents for the sample application.

\sdk\bin
Contains command line tools.

\sdk\docs
Contains documentation, such as this *Developer's Guide* in PDF format.

\sdk\include
Contains the header files.

\sdk\lib
Contains these library files:

trsagent.lib	client side library (static)
authverify.lib	server side library (import)
clientid.lib	client side library (import)
clientidd.lib	client side library (import)

NOTES:

- Clientid.dll must be installed in your client application path.
- Since authverify.lib is an import library, authverify.dll must be in your path specification to access the library.

\sdk\linux-i386
Contains authverify library for Linux* environment.

\sdk\src
Contains the \agent_test subdirectory, for testing purposes.

NOTE: In order to build the sample code, it is recommended that you have Microsoft Visual C++* Version 5.0 or newer.

Agent Installer

Agent Installer

This tool updates the agent database using the agents from the CD. This is part of the framework you need to provide (either by writing it yourself or by using Intel‘s separate Framework Reference Implementation product).

NOTE: Do not hard code the size of the agents, since that may change from release to release.

The config.txt file contains all of the information needed to use information from the CD. The config.txt syntax is as follows:

```
config::= {comment_line | blank_line | statement} EOF
comment_line::= # { chars } EOL
blank_line::= {whitespace} EOL
statement::= identifier {" "} = {" "} value EOL
identifier::= {a-z | A-Z | 0-9 | "_"}
value::= chars
chars::= {any ASCII character except EOL or EOF}
whitespace::={ ascii chars 0x09 - 0x0D or 0x20 }
EOL::= \n | \r\n
EOF::= {end of file character}
```

The file contains the following variables:

- VENDOR -- Intel (the standard vendor on all CDs produced)
- SERIAL_NUM -- Your Service ID, a unique identifier
- NUM_AGENTS -- The number of agents on the CD
- BUILD_DATE --The date the CD was produced
- CLIENTID_VERSION_MAJOR -- If the major version number has changed since your last installation, you may need to recompile your framework components)
- CLIENTID_VERSION_MINOR -- If the minor version number has changed, the new version should be backward compatible with your existing framework components)

- AUTHVERIFY_VERSION_MAJOR -- If the major version number has changed since your last installation, you may need to recompile your framework components)
- AUTHVERIFY_VERSION_MINOR -- If the minor version number has changed, the new version should be backward compatible with your existing framework components)
- REFRESH_RATE_IN_MINUTES -- Baseline of number of minutes an agent is used before expiring. This number may change in future releases, because it is based on security testing.

The Agent Installer should use the variables defined in the config.txt file to update the agent database with fresh agents. Some things to consider when writing the installer are:

- Verify the correct Service ID (SERIAL_NUM) from the CD, to ensure that you have gotten the correct CD for your particular service. This is especially important if you are running two or more discrete agent verification systems. Using a CD with the wrong service ID will result in failed authentication for registered users.
- Check the version numbers, both major and minor, to determine whether you need to recompile your source code.
- Determine the number of agents that are on the CD.
- Determine the refresh rate.
- Remember to allow for a smooth transition between two CDs. You will need to replace the agents with new ones (given the recommended refresh rate). Be sure that your installer tool will not mistake identically-named agents from two different CDs to be the same.

Agent Database

This component is accessed by both the Renewable Agent Scheduler and the Agent Installer components. Size of agents may vary from CD to CD, so do not hard code the size into your program.

Remember to allow for a smooth transition between two CDs, since you will need to replace the agents with new ones (given the recommended refresh rate). Suggested database fields you could implement: CD Serial Number; Agent Path; Start Time; Expire Time; Secret

NOTE: We recommend that you password-protect this database to ensure it is not readable by anyone other than the system administrators.

Renewable Agent Scheduler

This component communicates with the database and schedules a new agent every X minutes (where X is the refresh rate). The information the scheduler needs to know is: current agent (CD serial number plus agent number), start time of that particular agent, expire time, and which agent is next in line.

The numbering convention of agents on all CDs is auth0.dll through auth*n*.dll (where $n+1$ is the number of agents on that CD).

The recommended refresh rate for agents is in config.txt. For example, if the recommended rate were 20 minutes, a potential attacker would have at most 20 minutes to defeat that agent.

Client ID Database

This database stores information about existing accounts (registered users) and their client IDs. How this database is structured depends on your client application.

- If a single user has multiple machines, you need to store more than one Client ID per user.
- If a single machine may is used by multiple users, you need to store more than one Client ID per machine.

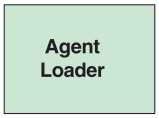
NOTE: The Client ID size may change in future releases of the Agents CD. Because of this, we recommend you do not hard code the Client ID size into your program. Also, legacy systems may be using data other than the Client ID, so size may vary.

Client ID

This component is part of the SDK. It communicates with the Client Application.
It is used only the first time a user logs in (Registration Process). Calling into this library generates a unique identifier for the client machine, which is then stored. The ClientID code is called from the client-side application.

IMPORTANT: Because this component is not tamper-resistant, it is highly recommended that a new user undergo the verification process (See "Agent Verification") after initially registering.

For example, this is the equivalent of a password-based system asking a new user to type a password and then making the user type it again to confirm it.



Agent Loader

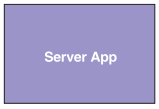
This component is part of the SDK, and is called from the client-side application. It is used as part of the verification process. It executes an agent, which results in an authentication code being generated.

For example, this is the equivalent of a password-based system asking an existing user to type a username and password to gain access.

Agent Verification

This component is part of the SDK. It is used for all subsequent logins after the initial registration (see "Agent Loader"). It compares the agent message with the stored data in the Client ID Database and either allows or denies access to the client.

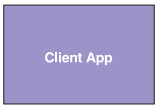
For example, this is the equivalent of a password-based system comparing the password a user types to a password file to determine if the user is allowed access.



Server Application

This is an existing component of your system, and refers to the server-side application you use currently to authenticate users.

To add the second factor of Machine-to-Machine Authentication, you need to store the Client ID data in your database in addition to your existing user data (such as a password). During the login process, the server application communicates with the Agent Scheduler and gets the current agent and secret. The agent is sent to the client side application, and computes the Client ID. It then returns to the server and compares the Client ID with the existing database.



Client Application

This is an existing component on the client side that is capable of communicating with the Agent Loader.

One example of a client-side application might be a plug-in for the user's Web browser.

Section 4: Sample Application

This section provides a detailed explanation of the agent_test.exe sample program provided in the SDK. This program demonstrates how various components of the SDK can be used in client-server application to perform more secure machine-to-machine authentication. However, to reduce code size and make the program easy to understand, we have combined both the client and server functionality into a single program. Source code for the application is located in the file src/agent_test/agent_test.c, which is reproduced in its entirety in "Appendix A" of this document.

Command Line Parameters

The Renewable Authentication Agent System contains the following command line tools to assist in development and deployment of a machine-to-machine authentication system.

insert_agent.exe

Agents shipped as part of the Intel Renewable Authentication System come in a packaged form. Packaged agents require the use of the trsagent.lib to load and execute the agents. The advantages of using this library are:

- Packaged agents are smaller and can be download over the Internet more quickly
- the loader library can check the signature of the agents before executing them, to ensure they are authentic.

However some implementations require the agents to be accessible through standard operating system calls. To enable this capability, we have included a utility called **insert_agent.exe** which inserts a packaged agent into a shell DLL to create a unique DLL agent. The tool can be used in a pre-processing step on the agents before inserting them into a database or sending them to clients. The usage of the tool is provided below.

SYNOPSIS

```
insert_agent.exe [<agentname.agent>] -s <shell.dll> -o <output.dll>
```

PARAMETERS

<agentname.agent>

Name of the "packaged" agent to insert. If not specified, the utility assumes the agent binary will be piped to the tool through STDIN.

-s

Name of shell DLL to insert agent into. The SDK comes with agent_shell.dll in the "sdk\bin" directory.

-o

Name of output DLL agent.

agent_test.exe / agent_testd.exe

There are actually two versions of this program, **agent_test.exe** and **agent_testd.exe**. **Agent_test.exe** should be used with Pentium III agents while **agent_testd.exe** should be used with debug agents provided in the SDK. Source code for this utility is provided in the SDK in the sdk/src/agent_test directory. When run, this tool does the following:

- Initializes parameters (e.g. service_id, secrets.txt, session_id)
- Collects the ClientID of the local machine. If **agent_testd.exe** is used, this value is based on a non-unique CPUID value rather than the processor serial number.
- Loads and executes the specified agent on the local machine to generate an authentication-code. If **agent_testd.exe** is used, only sample agents provided in the agent SDK should be specified (e.g. sampleXXd.agent) in the command line.
- Validates that the machine's ClientID matches the value indicated in the authentication code returned by the agent.

SYNOPSIS

```
agent_test<d> <auth.agent> -k <secrets.txt> [-n <secret_num>] -s <service.id>
```

PARAMETERS

<auth.agent>

Required name of agent to execute.

-k <secrets.txt>

File containing secret key. If not specified, "secrets.txt" is used by default.

-l

Label for secret in secret file. If not specified, the first key is used by default.

-s

File containing service_id. If not specified, "service.id" is used by default.

NOTES:

- The test utility is designed to work with both "packaged" agents (e.g. sampleXX.agent) as well as DLL agents (e.g. sampleXX.dll) generated with the "insert_agent.exe" utility.
- If you have problems running the application make sure all the required files are in the test utility's path. These include clientid.dll (or clientidd.dll), the agent being executed, service.id, authverify.dll, and secrets.txt.

Annotated Sample Code

Header Files

Some header files are only required by client-applications while others would be primarily used in server applications. Below is a list of header files included in this application, followed by a definition of each.

```
41. : #include <trsagent.h>
42. : #include <DSSTypes.h>
43. : #include <dsa_pub.h>
44. :
45. : #include <authagent.h>
46. : #include <authverify.h>
47. : #include <clientid.h>
```

trsagent.h: Defines the functions implemented in the TRSAgent.lib static library. Used by a client application to load and initialize packaged agents before executing them. Part of the initialization process involves checking the digital signature of the agent.

DSSTypes.h: Intel signs each agent using a 1024-bit DSA signature during the manufacturing process. This header file defines the data-structure used for DSA signatures, and is used primarily by client-side applications.

dsa_pub.h: Defines the functions implemented in dsa_pub.c. Currently, the source file contains a single function that returns the public DSA key used to sign all Intel Renewable Authentication Agents. Used primarily by client-side applications.

authagent.h: Defines the return codes and prototypes of the functions implemented in the agents. Used by the client application during the verification process to generate an authentication code.

clientid.h: Defines the return codes and prototypes of the functions implemented in clientid.dll. Used during the registration process to compute the ClientID of a client-system.

NOTE: If agents are sent to the client application in the form of a DLL, the header files **trsagent.h**, **dsstypes.h**, and **dsa_pub.h** are not required. However, DLL agents tend to be larger in size. Using the operating system to load a DLL agent does not provide any signature checking of the agents. Therefore, if you use the DLL interface, you must ensure that it is verified before using it.

Helper Functions

The sample program contains several helper functions to parse parameters provided in the various configuration files on the Renewable Authentication Agent CD. In an actual server implementation, these functions would normally be implemented in an installer program. The two primary helper functions are:

```
146. : char *read_serviceidfile(const char *file,int *length)

187. : char *read_secretfile(const char *file,const char *secret_label, int
secret_size)
```

The function **read_serviceidfile(…)** is used to parse the service.id file provided on the CD. The contents of the service.id file is a variable length string that should be interpreted as binary bytes (e.g. “12232233 56552434 68818732 82544694” or “serviceid”) . This value is required during the registration on the client and the verification on the server. This **read_serviceidfile(…)** function calls into the following sub-functions to help parse the secrets file:

```
94. : static int count_bytes(FILE *fp)
```

which returns the size of the file.

The function **read_secretfile(…)** is used to parse the secrets.txt file provided on the CD. Each agent has a secret embedded in it, which should only be known to the server. The secret for each agent provided on a CD is encoded as hexadecimal digits with a label. E.g.:

```
# Sample secrets used by both the debug and katmai agents provided

SECRET 0:
3823ca98 b461995f 6a92982a 51942daf 40136ad8 ea36db0b ddc0e98e b89c58a3
67ae396e dbe1ff86 a52f31ac 4f34e740 4d5000fc 426e277d 2f923d80 68f50eed
ae206a32 4be6cac9 b77f198c b561b95b 0ab0de4b c19a5ddc 28e546f1 7d394865
91a80cf7 151494ad b76ce46b e92fa4fc b655fc42 25449f8f d189cc96 43c2d5c6
e5736ceb 0ab5380e c694c96c a1e37ad2 187c16f1 96684066 6ba9b29c e09144a7
eb239527 46a70613 58f36f49 ba0b6ff4 9992d02e 6a59f592 1b33867e ed046217
709be587 fb125526 8b1f0414 078ab98e 52b0b89b 5a2635f6 21806c35 62d81cab
3b52b0b3 a3865e39 3c16a5fd 27393ba0 7d68ac49 a059c513 bbcac324 02f671cb
bbe5ce68 144dd090 8e723160 96b74601 9eb83694 5abecdd7 f0e84745 79d14ba7
```

```
fc366664 6b663581 6218ac60 cb6a07fa 81fa739a b9c5ee58 71665822 b066a487
4324e880 a8674df2 aaacc30b 9c72c2a0 c24f382a 6591ef1b f7543a98 c91bcd43
65b0cf8a bde29121 55471737 cdc7087e 766002c3 f0a83845 acd781af 6ca00ef1
3f82f1a3 c56e040a 66caaadd f78189ed e717bed7 7fa25787 73c7e340 89e6ddbc
50dbbb45 d4f6b42e 6c2465e2 8dd8dc4b 1ae7dae6 1a3ba397 842d9dab 9a058784
b12d06a8 b0c5f2cd cb478b14 f11f7edf 8dbea963 7b03cf88 faa8321d e7077930
7bff04c3 416f980c 27ad19ac 449d5e3d b5c5c692 c4650b5f 75850131 27cb94d5

SECRET 1:
5d118581 224ce044 71fb8ca0 8ae77a31 b04388f4 6ddc9148 0bd1b38a 1bf6d298
2e72490e 530e28f7 91343c9f ea74b1e2 9ff022f5 d4bf5a6c 58a3f9ed be886d21
8d1cce23 c64257ff 7b383b79 d331a2d4 18cbd9e1 8c8ecfd0 3b7595a0 1f35bc5c
3c5e7dca 4889ba8c 2f020e3a c1ee5bfb 249b47b2 8e2ea757 fc6cc373 4b48437d
31acd34a bc493248 11662831 a3c64894 2c8fb480 2f942668 9d267979 001aaccf
.
.
.
```

NOTE: The secrets.txt file is encrypted for production agents and must be decrypted before this function can work. This **read_secretfile(…)** function calls into the following sub-functions to help parse the secrets file:

```
61. : int is_comment(char *line)
```

Returns 1 if current line is a comment. Returns 0 otherwise.

```
67. : int is_empty(char *line)
```

Returns 1 if current line is empty. Returns 0 otherwise.

```
78. : int is_label(char *line)
```

Returns 1 if current line is empty. Returns 0 otherwise.

```
84. : int get_label(char *line)
```

Returns a pointer to the secret # if line is a label. Returns 0 otherwise.

```
123. : int xlate_line(const char *line, char *dest, int dest_size, int offset)
```

Converts the hexadecimal digits contained in the secrets.txt file into a binary representation.

Application Initialization

The **main(…)** function is where most of the work is accomplished. It begins by initializing the variables used in the rest of the program.

```
339. : int main(int argc, char *argv[])
340. : {
341. :   DSSPublicKey publicKey;
342. :   char *agent_name=NULL;
.
.
.
```

The pseudo random generator is then initialized. It is used by the server application to generate a rarely repeating session id for verification.

```
363. :   /* Initialize the non-crypto pseudo random number generator. */
364. :   srand(time(NULL));
365. :
```

Finally the program calls into **get_params(…)** which simply parses the command line parameters and returns the values of the secret filename, agent filename, secret label, service id filename, and debug flags to the main program.

```
366. :   if (!get_params(argc, argv, &agent_name, &secret_file, &secret_num,
367. :            &service_file, &const_debugauth))
368. :     {
.
.
```

.

Registration

Server-Side Initialization
Registration is the process by which the identity of a remote machine is determined and access to content or service is generated. This SDK primarily deals with the first portion of this problem: identifying the remote machine.

Before registration can occur, the server must pass its service id to the client system. The client uses this value to generate a ClientID that will match the value embedded in the agents sent from that server. In the sample application, we simply load the service id from the file.

```
393. :   /* Get the service id.  A server would know this value for all agents. */
394. :   printf("- Loading the agent's service file '%s'\n", service_file);
395. :   service = read_serviceidfile(service_file,&serviceid_length);
396. :   if(!service)
397. :     {
398. :       fprintf(stderr, "%s: unable to read service file\n", argv[0]);
399. :       return 1;
400. :     }
401. :
```

Client-Side Registration
Once the client has received the service id from the server, the client calls in to the clientid dll to generate a ClientID for the remote machine. This returned value is then sent back to the server.

```
412. :   /* Get the client id.  A server would retrieve this value for
413. :      registered users. */
414. :   printf("- Calling GetClientID\n");
415. :   clientid_length = GetClientID(GET_CLIENTIDLENGTH);
416. :   if(clientid_length <= 0)
417. :   {
418. :       fprintf(stderr, "%s: unable to get client id length ERR=%d\n",
419. :        argv[0], clientid_length);
420. :       return 1;
421. :   }
422. :
423. :   clientid = malloc(clientid_length);
424. :   if(clientid == NULL)
425. :   {
426. :       fprintf(stderr, "%s: out of memory\n", argv[0]);
427. :       return 1;
428. :   }
429. :
430. :   err = GetClientID(GET_CLIENTID, service, serviceid_length, clientid);
431. :   if(err != 1)
432. :   {
433. :       fprintf(stderr, "%s: unable to get client id ERR=%d\n",
434. :        argv[0], err);
435. :       return 1;
436. :   }
437. :
```

NOTE: The client application should query for the length of the Client ID and accommodate for a variable length return value. This flexibility is recommended for ease of integration of possible future extensions to the SDK that would allow additional machine identifiers other than PS# (e.g. legacy PC fingerprinting).

Server-Side Registration
The final step of the registration process is the storage of the Client ID into a database or access control list on the server. The sample application does not perform this step since verification is done immediately. Note that since registration does not use tamper-resistant components, the verification step should be integrated with registration process, if access to content or services will be issued immediately.

Verification

Server-Side Initialization
To perform verification, the server application must first generate a session id for the transaction. The session id is used to prevent replay attacks on the authentication code generated by an agent, and therefore should be a very rarely repeating string. Once generated, the session id must be passed to the agent as a parameter on the client system.

```
379. :   /* Pick a session id.  This should be unique for every call to the
380. :      agent: a monotonic counter would work.
381. :
382. :      Here, we use a poor pseudo random number to get 128 bits.  This
383. :      is also sufficient to hinder most attacks. */
384. :
385. :   printf("- Choosing session identifier\n");
386. :   session[0] = '\0';
387. :   if(!const_debugauth)
388. :   {
389. :     for(i = 0; i < SESSION_ID_SIZE-1; i += 2)
390. :       sprintf(session+i, "%2.2x", rand() & 0xff);
391. :   }
392. :
```

In addition to generation of a session id, the server must also retrieve the current agent, based on a predetermined scheduling policy. In this sample application, the current agent is supplied through a command line parameter.

The server application then retrieves the secret for the current agent from a database. In the sample application, this value is retrieved directly from the secrets.txt file.

```
402. :   /* Load the agent's secret.  Each agent has its own
403. :      secret. */
404. :   printf("- Loading the agent's secret '%s'\n", secret_file);
405. :   key = read_secretfile(secret_file,secret_num,secret_bytes);
406. :   if(!key)
407. :     {
408. :       fprintf(stderr, "%s: unable to find agent secret\n", argv[0]);
409. :       return 1;
410. :     }
411. :
```

Finally, the server sends both the agent and session id to the client machine and waits for an authentication code to return. The server must cache the session id and secret for the duration of the transaction.

Client-Side Verification
Once the agent and the session id is downloaded to the client, the client application must load, initialize, and execute the agent.

As part of initialization, the agent loader library verifies the signature of the agent. The client application must provide the loader the root signing public-key for the agents that it trusts. The key for our current SDK is provided in the dsa_pub.c file. To retrieve the key, the sample application calls to get_public_key().

```
438. :   /* Load Authentication agent */
439. :   printf("- Loading authentication agent '%s'\n",agent_name);
440. :
441. :   get_public_key(&publicKey);
```

Next, the sample application attempts to load the agent using the agent loader library. This function determines if the agent is a valid packaged agent, verifies the digital signature, and returns a handle to the agent.

```
442. :   st = load_agent(agent_name, &publicKey, &hAgent);
```

If the load succeeds, the agent entry point is retrieved.

```
443. :   if(st == TRSAGENT_OK) {
444. :
445. :     /* Since packed agents are marked with a 128-bit universally
446. :        unique ID, there's no real chance that this isn't an agent by
447. :        accident. */
448. :
449. :     /* Get the agent's entry point */
450. :     agentf = (AUTHENTICATION_AGENT) get_agent_proc(hAgent);
451. :     if (agentf == 0) {
452. :       fprintf(stderr,
453. :        "%s: unable to get entry point for %s\n",
454. :        argv[0], agent_name);
455. :       return 1;
456. :     }
457. :   }
```

If the load fails, the application assumes the agent passed in was a DLL agent and uses the Win32 LoadLibrary call to load the agent. Note that this technique does not perform any signature checking on the agent. An actual client-server system would either use packaged or DLL agents, but not both. As a result, this code would not be required.

```
458. :   else if(st == TRSAGENT_INVALID_AGENT) {
459. :
460. :     /* Failed to load as a packed agent, try it as a DLL.  */
461. :     hDLLAgent = LoadLibrary(agent_name);
462. :     if (!hDLLAgent) {
463. :       fprintf(stderr, "%s: unable to load authentication agent DLL\n",
464. :        argv[0]);
465. :       return 1;
466. :     }
467. :
468. :     /* Get the agent's entry point. */
469. :     agentf = (AUTHENTICATION_AGENT) GetProcAddress(hDLLAgent, "agent");
470. :     if (!agentf) {
471. :       fprintf(stderr,
472. :        "%s: unable to find entry point in authentication agent DLL\n",
473. :        argv[0]);
474. :       return 1;
475. :     }
```

If the load fails, the client application checks error conditions and post an error code back to the server. In our sample application, we simply print an error and exit. The error codes are defined in authagent.h."

```
476. :   }
477. :   else /* It really did fail badly... */
478. :     {
479. :       char *str = NULL;
480. :       switch(st)
481. : {
482. : case TRSAGENT_ALLOC_FAILED:   /* Allocation of memory failed */
483. :   str = "%s: Out of memory\n";
484. :   break;
485. : case TRSAGENT_DSS_VERIFY_FAILED:/* DSS Signature verification failed */
486. :   str = "%s: DSS Signature verification failed\n";
487. :   break;
488. : case TRSAGENT_FILE_LOAD_FAILED:
489. :   str = "%s: File load failed\n";
490. :   break;
491. : case TRSAGENT_DSS_INVALID_KEY:       /* Invalid DSS Key was passed in */
492. :   str = "%s: Attemped to use invalid DSS key\n";
493. :   break;
494. : default:
495. :   str = "%s: Unknown error code: %d\n";
496. : }
497. :       fprintf(stderr, str, agent_name, st);
498. :       return 1;
499. :     }
500. :
```

Once the agent is loaded and initialized, the client application calls into the agent to generate an authentication code. The application first queries the agent for the length of the authentication code and

accommodates variable length codes. This is recommended to provide a migration path to future enhancements to the agents.

```
501. :   /* Call the authentication agent. */
502. :   printf("- Calling authentication agent\n");
503. :
504. :   auth_length = agentf(GET_AUTHLENGTH);
505. :   if(auth_length <= 0)
506. :   {
507. :       fprintf(stderr, "%s: unable to get authentication length\n",
argv[0]);
508. :       return 1;
509. :   }
510. :
511. :   auth = malloc(auth_length);
512. :   if(auth == NULL)
513. :   {
514. :       fprintf(stderr, "%s: out of memory\n", argv[0]);
515. :       return 1;
516. :   }
517. :
518. :   if(const_debugauth)
519. :       err = agentf(GET_CONST_DEBUG_AUTHENTICATION, session, auth);
520. :   else
521. :       err = agentf(GET_AUTHENTICATION, session, auth);
522. :
523. :   if(err != 1)
524. :   {
525. :       fprintf(stderr, "%s: authentication agent failed ERR=%d\n",
526. :        argv[0], err);
527. :       return 1;
528. :   }
529. :
```

Production agents generate a different authentication code on every transaction, even if a constant session id is passed in. Using GET_CONST_DEBUG_AUTHENTICATION always returns a constant authentication code for a particular machine (assuming a constant session id). This functionality was found useful in debugging client-server applications. Once the agent is executed, the returned authentication code is sent back to the server application to be verified.

NOTE: The GET_CONST_DEBUG_AUTHENTICATION action is only enabled on the debug agents provided in the SDK.

Server-Side Verification

Once the server has received the authentication code from a remote machine, the server must validate the authentication code against an expected ClientID collected during registration. This would generally take the form of a database lookup. In our sample application, registration and verification were combined in a single execution pass.

NOTE: The ValidateAuthentication function requires the cached values of the agent secret (key) and the session id.

```
530. :   /* Print the authentication code */
531. :   for(i = 0; i < auth_length; i += sizeof(long))
532. :       printf(" %08x", *(long*)(auth+i));
533. :
534. :   printf("\n- Validating authentication code\n");
535. :
536. :   if(ValidateAuthentication(auth, clientid, session, key))
537. :       printf("Validation PASSED\n");
538. :   else
539. :       printf("Validation FAILED\n");
```

Upon successful return, the server application grants access to the requested information or service.

Application Cleanup

This application terminates after cleaning up any allocated buffers generated through its execution. Similarly, a server application needs to clean up any data allocated for a particular transaction.

```
540. :
541. :   free(clientid);
542. :   clientid = NULL;
543. :
544. :   if(auth != NULL)
545. :   {
546. :       free(auth);
547. :       auth = NULL;
548. :   }
549. :
550. :   if(hAgent)
551. :     free_agent(hAgent);
552. :
553. :   if(hDLLAgent)
554. :     FreeLibrary(hDLLAgent);
555. :
556. :   free(service);
557. :   free(key);
558. :   key = NULL;
559. :   return 0;
560. : }
```

Section 5: API Specifications

This section of the document provides a detailed specification for the programming interfaces provided in the SDK. The interfaces described in order are:

- **ClientID API:** Calculates an identifier called a ClientID of a client machine based on its processor serial number. This module is not tamper-resistant and the results should not be trusted without the subsequent use of an agent to verify the results.
  get_clientidlength
  get_clientid

- **TRSAgent API:** Initializes a "packaged" agent and retrieves its entry point. Packaged agents should be used when bandwidth to the client is low and minimizing transaction speed is critical. Please refer to the "Agents Tools" section for more details on the differences between packaged and DLL agents.
  get_public_key
  use_agent
  load_agent
  get_agent_proc
  free_agent

- **Agent API:** After an agent is loaded and initialized, this API calls into the agent to generate an "authentication code". For packaged agents, the entry point to this API can be retrieved through the TRSAgent API described above. For DLL agents, the entry point can be retrieved using the standard Microsoft Windows* System Calls: LoadLibrary() and GetEntryPoint()
  get_authlength
  get_authentication
  get_const_authentication

- **AuthVerify API:** Used by the server application to verify the authentication code against the expected ClientID for a particular machine.
  ValidateAuthentication

8.1 ClientID API

Source File: None Header File: clientid.h Lib: clientid.lib DLL: clientid.dll

NAME

GetClientID

SYNOPSIS

```
int GetClientID(<action>...
```

DESCRIPTION

This function is used to calculate an identifier of a client machine (ClientID) based on its processor serial number. Depending on which of the variable arguments (actions) is used with the function, it: 1) determines the amount of memory required for the client id buffer, and 2) calculates the ClientID. This module is not tamper-resistant and the results should not be trusted without the subsequent use of an agent to verify the results.

PARAMETERS

<action>

Can be either of these two values: GET_CLIENTIDLENGTH, GET_CLIENTID. See the separate APIs that follow for the parameters of each action.

RETURN VALUE

This function returns an integer. Depending on which of the two actions is selected, the return value is different. See the following APIs for return value specific to each action.

ERROR CODES

CLIENTID_INVALID_ACTION	Action not supported

SEE ALSO

agent

8.1.1 GetClientID(GET_CLIENTIDLENGTH)

SYNOPSIS

```
int GetClientID(GET_CLIENTIDLENGTH)
```

DESCRIPTION

When used with the action GET_CLIENTIDLENGTH, this function determines how much memory to allocate before getting client id.

PARAMETERS

service_id

A binary buffer containing the service_id to use in generating the client identifier. The service identifier must match the value embedded in the authentication agents.

service_id_length

The length of the service_id buffer.

RETURN VALUE

Calling the agent with the GET_CLIENTIDLENGTH returns the CLIENTID_SIZE.

ERROR CODES

CLIENTID_INVALID_ACTION	Action not supported

SEE ALSO

agent

8.1.2 GetClientID(GET_CLIENTID, ...)

ACTION

GET_CLIENTID

SYNOPSIS

```
int GetClientID(GET_CLIENTID,
                char *service_id,
                int service_id_length,
                char *id)
```

DESCRIPTION

Using this function with the action GET_CLIENTID computes an identifier called a ClientID (Hash(PS# , ServiceID)) of a client machine based on the processor serial number of the machine the application is executing on. This module is not tamper-resistant and the results should not be trusted without the subsequent use of an agent to verify the results.

PARAMETERS

service_id

A binary buffer containing the service_id of the service into which the client is requesting authentication. The service identifier must match the value embedded in the authentication agents sent from that service.

service_id_length

The length of the service_id buffer.

id

Upon successful completion, id contains an identifier for the client machine. The function assumes the buffer passed in is exactly the length returned by GetClientID(GET_CLIENTIDLENGTH).

RETURN VALUE

This function returns an integer. Given the proper service identifier, it computes the hash (Processor Number, Service) and returns CLIENTID_OK if successful, or an error code if not successful.

ERROR CODES

CLIENTID_PN_OFF	Processor number feature is turned off
CLIENTID_NO_PN	No processor number feature available
CLIENTID_INVALID_ACTION	Action not supported

SEE ALSO

agent

8.2 TRS Agent API

8.2.1 get_public_key

Source File: dsa_pub.c Header File: dsa_pub.h Lib: none DLL: none

NAME

get_public_key

SYNOPSIS

```
int get_public_key(DSSPublicKey *key)
```

DESCRIPTION

The source file dsa_pub.c implements this function. The function retrieves the public-key of the DSA key-pair used to sign all Intel Authentication Agents. The returned value should be passed as a parameter to use_agent or load_agent so that the integrity of the agents can be verified before they are executed. This protects the client machine from executing a rogue agent.

PARAMETERS

key

the pointer to the data structure that is the Public Key.

RETURN VALUE

If the function was succesful, it returns an integer.

SEE ALSO

use_agent
load_agent

8.2.2 use_agent

Source File: none Header File: trsagent.h Lib: trsagent.lib DLL: none

NAME

use_agent

SYNOPSIS

```
int use_agent(unsigned char *buffer,
                          int length,
                          DSSPublicKey *publicKey, AGENT_HANDLE *pAgent)
```

DESCRIPTION

This function is used to initialize a "packaged" agent and retrieve its entry point. This is one of two different ways to initialize an agent, the other being load_agent. The use_agent function is used if the agent is already in a memory buffer on the client machine, whereas load_agent is used if the agent resides in the file system and must be loaded from a file.

PARAMETERS

buffer

Input that specifies location of agent.

length

Input that specifies size of buffer.

publicKey

Input that verifies that agent is a valid Intel Authentication Agent.

pAgent

Output that verifies the signature on the agent.

RETURN VALUE

This function returns an integer. If the function was successful, it returns TRSAGENT_OK. Otherwise, it returns an error code.

ERROR CODES

TRSAGENT_INVALID_AGENT	Invalid agent interface
TRSAGENT_ALLOC_FAILED	Allocation of memory failed
TRSAGENT_DSS_VERIFY_FAILED	DSS signature verification failed
TRSAGENT_DSS_INVALID_KEY	Invalid DSS key was passed in

NOTE

It is recommended that use_agent is used instead of load_agent whenever possible. Storing the agent on the disk provides an opportunity for attackers to replace the stored file with their own.

SEE ALSO

load_agent

8.2.3 load_agent

Source File: none Header File: trsagent.h Lib: trsagent.lib DLL: none

NAME

load_agent

SYNOPSIS

```
int load_agent(char *file_name,
               DSSPublicKey *publicKey,
               AGENT_HANDLE *pAgent)
```

DESCRIPTION

This function is used to initialize a "packaged" agent and retrieve its entry point. This is one of two different ways to initialize an agent, the other being use_agent. Load_agent is used if the agent resides in the file system and must be loaded from a file, whereas use_agent is used if the agent is already in a buffer on the client machine

PARAMETERS

file name

Input that specifies location of agent.

publicKey

Input that verifies that agent is an official Intel agent.

pAgent

Output that verifies the signature on the agent.

RETURN VALUE

This function returns an integer. If the function was successful, it returns TRSAGENT_OK. Otherwise, it returns an error code.

ERROR CODES

TRSAGENT_INVALID_AGENT	Invalid agent interface
TRSAGENT_ALLOC_FAILED	Allocation of memory failed
TRSAGENT_DSS_VERIFY_FAILED	DSS signature verification failed
TRSAGENT_FILE_LOAD_FAILED	Loading agent file was unsuccessful
TRSAGENT_DSS_INVALID_KEY	Invalid DSS key was passed in

NOTE

It is recommended that use_agent is used instead of load_agent whenever possible. Storing the agent on the disk provides an opportunity for attackers to replace the stored file with their own.

SEE ALSO

use_agent

8.2.4 get_agent_proc

Source File: none Header File: trsagent.h Lib: trsagent.lib DLL: none

NAME

get_agent_proc

SYNOPSIS

```
AGENT_PROC get_agent_proc(AGENT_HANDLE agent)
```

DESCRIPTION

This function returns a pointer to the agent entry point. The agent can then be authorized.

PARAMETERS

agent

agent handle (the result of the load_agent or use_agent output).

RETURN VALUE

This function returns a pointer to the agent.

NOTE:

This function is required before the agent can be executed.

8.2.5 free_agent

Source File: none Header File: trsagent.h Lib: trsagent.lib DLL: none

NAME

free_agent

SYNOPSIS

```
void free_agent(AGENT_HANDLE agent)
```

DESCRIPTION

This function frees and wipes (deletes securely) the memory that was used by the agent. It is symmetrical with use_agent or load_agent, and should always be run as a final function to cleanup. This function should be run immediately after retiring an agent.

PARAMETERS

agent

agent handle.

RETURN VALUE

There is no return value.

SEE ALSO

use_agent
load_agent

8.3 Agent Interface

Source File: None Header File: authagent.h Lib: None DLL: authXX.dll (generated using insert_agent utility) Modules: sampleXX.agent or authXX.agent

NAME

agent

SYNOPSIS

```
int agent(<action>...
```

DESCRIPTION

This function can be used in two ways, depending on the <action> parameter passed into the agent. First, the agent can be called with the GET_AUTHLENGTH action to determine the amount of memory to be allocated for the authentication message. Second, the agent can be called with the GET_CONST_AUTHENTICATION action to execute the agent on a client machine and generate an authentication code.

PARAMETERS

<action>

Can be any one of these three values: GET_AUTHLENGTH, GET_AUTHENTICATION, or GET_CONST_AUTHENTICATION. See the separate API's that follow for parameters of each action.

RETURN VALUE

This function returns an integer. Depending on which of the two actions is selected, the return value is different. See the following APIs for return value specific to each action.

ERROR CODES

AUTH_INVALID_ACTION Action not supported

8.3.1 agent(GET_AUTHLENGTH)

SYNOPSIS

```
int agent(GET_AUTHLENGTH)
```

DESCRIPTION

The GET_AUTHLENGTH action returns the buffer length of the “authentication code” generated by this agent.

PARAMETERS

None.

RETURN VALUE

If the function was successful, returns the required buffer length to the GET_AUTHENTICATION function. Otherwise, it returns an error code.

ERROR CODES

AUTH_INVALID_ACTION	Action not supported

NOTE

Error codes for this function will always be an integer less than zero.

8.3.2 agent(GET_AUTHENTICATION, ...)

SYNOPSIS

```
int agent(GET_AUTHENTICATION,
          char *session_id,
          char *auth)
```

DESCRIPTION

Executes the tamper resistant agent and generates an authentication code for the machine that the applications is running on. The authentication code can then be sent to a server where it can be verified against a database of authorized ClientIDs.

PARAMETERS

session_id

A NULL terminated ASCII string that contains a "rarely-repeating" string. The recommended value is the ASCII representation of a 32-bit counter (e.g. "234234"). Changing the session id on every transaction ensures that an attacker discovering an old authentication code cannot replay the message to gain access to the server.

auth

Upon successful completion, contains the authentication code returned by the agent. Assumes that the buffer is at least the length returned by agent (GET_AUTHLENGTH).

RETURN VALUE

This function returns an integer. If the function was successful, it returns AUTH_OK. Otherwise, it returns an error code.

ERROR CODES

AUTH_PN_OFF	Processor number feature is turned off
AUTH_NO_PN	No processor number feature available
AUTH_INVALID_SESSIONID	SessionID parameter invalid

SEE ALSO

get_agent_proc

8.3.3 agent(GET_CONST_DEBUG_AUTHENTICATION, ...)

SYNOPSIS

```
int agent(GET_CONST_AUTHENTICATION,
          char *session_id,
          char *auth)
```

DESCRIPTION

The GET_CONST_DEBUG_AUTHENTICATION action is only implemented in the sample debug agents provided in the SDK. This function is used to generate a constant authentication code for a particular machine assuming a constant session_id and is useful in prototyping or debugging a client-server system.

PARAMETERS

session_id

A null terminated ASCII string that contains a "rarely-repeating" string. The recommended value is the ASCII representation of a 32-bit counter (e.g. "234234"). Changing the session id on every transaction ensures that an attacker discovering an old authentication code can not replay the message to gain access to the server.

auth

Upon successful completion, contains the authentication code returned by the agent. Assumes that the buffer is at least the length returned by agent (GET_AUTHLENGTH).

RETURN VALUE

This function returns an integer. If the function was successful, it returns AUTH_OK. Otherwise, it returns an error code.

ERROR CODES

AUTH_PN_OFF	Processor number feature is turned off
AUTH_NO_PN	No processor number feature available
AUTH_INVALID_SESSIONID	SessionID parameter invalid

NOTES

Production agents do not implement GET_CONST_DEBUG_AUTHENTICATION.

SEE ALSO

get_agent_proc

8.4 AuthVerify API

8.4.1 ValidateAuthentication

Source File: Header File: authverify.h Lib: authverify.lib DLL: authverify.dll

NAME

ValidateAuthentication

SYNOPSIS

```
int ValidateAuthentication(
                        const char *auth_code,
                        const char *client_id,
                        const char *session_id,
                        const char agent_secret[AGENT_SECRET_LENGTH])
```

DESCRIPTION

This function compares an authentication code returned by an agent with a single ClientID. Usually, the ClientID will represent a machine that was registered is authorized to gain access to the service.

PARAMETERS

auth code

A binary buffer containing the value returned from the authentication agent on the client system. The buffer is assumed to be at least the length returned by agent(GET_AUTHLENGTH).

client_id

A binary buffer containing the value returned from GetClientID ( ). The buffer is assumed to be at least the length returned by GetClientID(GET_CLIENTIDLENGTH). This value should be stored on the server after registering the user.

session_id

A NULL terminated ASCII string chosen by the server for this session. This value must match the session identifier supplied when calling the authentication agent. This also must be stored on the server during the session. The recommended value is the ASCII representation of a 32-bit counter (e.g. "234234").

agent_secret

Each agent has a secret value embedded in the agent and is provided in plain-text to the server. This buffer is assumed to be AGENT_SECRET_LENGTH long. The actual secrets should be protected on the server since the security of agents depends on an attacker not knowing this value.

RETURN VALUE

This function returns an integer. If the function was successful, it returns VALIDATION_OK. Otherwise, it returns an error code.

ERROR CODES

VALIDATION_FAILED Authentication code did not match clientid

Appendix A: Agent_Test.c Source Code

```
1. : /**
2. : ***  Copyright  (C) 1998-2007 Intel Corporation. All rights reserved.
3. : ***
4. : *** The information and source code contained herein is the exclusive
5. : *** property of Intel Corporation and may not be disclosed, examined
6. : *** or reproduced in whole or in part without explicit written authorization
7. : *** from the company.
8. : **/
9. :
10. : /* agent_test.c
11. :
12. :    This program demonstrate how to load and execute an authentication
13. :    agent and validate the returned authentication code.
14. :
15. :    The agents can be either packed agents as provided on the CD, or
16. :    inserted into a DLL.  Methods for addressing both types are
17. :    provided here.
18. :
19. :    Use the "insert_agent.exe" utility to insert agents into DLLs.
20. :
21. :    Usage: agent_test <auth.agent> -k <secret.txt> [-n <secret_num>]
22. :         -s <service.id>
23. :
24. :   -k: File containing secret key.
25. :   -l:  Label for secret in secret file.
26. :           If not specified, first key is used
27. :   -s: File containing service_id.
28. :
29. : */
30. :
31. : #include <windows.h>
32. : #include <stdio.h>
33. : #include <stdlib.h>
34. : #include <time.h>
35. : #include <errno.h>
36. : #include <string.h>
37. : #include <ctype.h>
38. : #include <sys/types.h>
39. : #include <sys/stat.h>
40. :
41. : #include <trsagent.h>
42. : #include <DSSTypes.h>
43. : #include <dsa_pub.h>
44. :
45. : #include <authagent.h>
46. : #include <authverify.h>
47. : #include <clientid.h>
48. :
49. : /* In this sample, SESSION_ID_SIZE is four 32-bit numbers in hex,
50. :    followed by a '\0'. */
51. : #define SESSION_ID_SIZE (4*8+1)
52. :
53. : /* Used by load_hexfile().  This is the maximum line length before a
54. :    newline. */
55. : #define LINE_LENGTH 1024
56. :
57. : /* This is the entry point to the authentication agent found in
58. :    agent_shell.dll */
59. : extern int agent(int func, ...);
60. :
61. : int is_comment(char *line)
62. :      /* Check to see if this line is a comment */
63. : {
64. :   return (line[0] == '#');
65. : }
66. :
67. : int is_empty(char *line)
68. :      /* Check whether this line is empty */
69. : {
70. :   int i;
71. :   for(i = 0; line[i] != '\0'; i++)
72. :     if(!isspace(line[i]))
73. :       return 0; /* nope, not empty */
74. :
75. :   return 1; /* empty */
76. : }
```

```
77. :
78. : int is_label(char *line)
79. :      /* returns 1 if line is a secret label, 0 otherwise */
80. : {
81. :   return (strncmp(line,"SECRET",6)==0);
82. : }
83. :
84. : int get_label(char *line)
85. :      /* returns label for secret */
86. : {
87. :   int p=0;
88. :   if (sscanf(line,"SECRET %d:",&p)==1) {
89. :     return p;
90. :   }
91. :   return -1;
92. : }
93. :
94. : static int count_bytes(FILE *fp)
95. :      /* Found out how many bytes are given in the file fp, or return -1
96. :        on error. */
97. : {
98. :   char line[LINE_LENGTH];
99. :   int count;
100. :
101. :    count = 0;
102. :
103. :    fgets(line, LINE_LENGTH, fp);
104. :    while(!ferror(fp) && !feof(fp))
105. :      {
106. :        if(!is_comment(line) && !is_empty(line))
107. :          {
108. :     size_t i;
109. :     for(i = 0; i < strlen(line); i++)
110. :               {
111. :         if(isxdigit(line[i]))
112. :           count++;
113. :             }
114. :          }
115. :        fgets(line, LINE_LENGTH, fp);
116. :      }
117. :    if(!ferror(fp))
118. :      return count / 2;
119. :
120. :    return -1;
121. : }
122. :
123. : int xlate_line(const char *line, char *dest, int dest_size, int offset)
124. :      /* Translate the ascii hex digits in line.  Returns number of
125. :  digits set in dest. */
126. : {
127. :   int i, /* index into line */
128. :     j; /* index into dest */
129. :
130. :    for(i = 0, j = offset; (offset<dest_size) && line[i] != '\0'; i++)
131. :      {
132. :        if(isxdigit(line[i]) && isxdigit(line[i+1]))
133. :          {
134. :     int digit;
135. :     sscanf(line+i, "%2x", &digit);
136. :     i++;
137. :
138. :     dest[j] = digit;
139. :     j++;
140. :
141. :          }
142. :      }
143. :    return (j-offset);
144. : }
145. :
146. : char *read_serviceidfile(const char *file,int *length)
147. :      /* Loads and reads the specified session.id file and returns it
148. :  contents in a buffer.
149. :
150. :  On success, returns a pointer to the loaded binary data &
151. :  length is set to length of buffer.
152. :
153. :  Returns NULL otherwise & length is set to 0.
154. :  free() should be used to free the allocated memory.
155. :      */
156. : {
157. :   char *service=NULL;
```

```
158. :   unsigned int slength=0;
159. :   struct _stat st;
160. :   FILE *fservice=NULL;
161. :   *length=0;
162. :
163. :   if(_stat(file, &st) != 0) {
164. :     return NULL;
165. :   }
166. :   slength=st.st_size;
167. :
168. :   if (!(fservice=fopen(file,"r"))) {
169. :     return NULL;
170. :   }
171. :
172. :   if (!(service = (char *) malloc(slength))) {
173. :     fclose(fservice);
174. :     return NULL;
175. :   }
176. :
177. :   if (fread(service,1,slength,fservice)!=slength) {
178. :     free(service);
179. :     fclose(fservice);
180. :     return NULL;
181. :   }
182. :   fclose(fservice);
183. :   *length=slength;
184. :   return service;
185. : }
186. :
187. : char *read_secretfile(const char *file,const char *secret_label, int secret_size)
188. :
189. :      /* Loads and reads the given file.  It should contain a list of hex
190. :         digits with optional white space and comments.  Assumes text file
191. :         with max line length< LINE_LENGTH. Assumes file ends with '\n'
192. :
193. :    On success, returns a pointer to the loaded binary hex data.
194. :                                  Returns NULL otherwise.
195. :        free() should be used to free the allocated memory.
196. :  */
197. : {
198. :   FILE *fp=NULL;
199. :   char *bytes=NULL;
200. :   int offset=0;
201. :   char line[LINE_LENGTH];
202. :   char *label=NULL;
203. :   int found=FALSE;
204. :   int secretIndex=0;
205. :
206. :   fp = fopen(file, "r");
207. :   if(!fp) {
208. :     return NULL;
209. :   };
210. :
211. :
212. :   /* Get the file size and allocate enough memory for the bytes */
213. :   fseek(fp, 0L, SEEK_SET);
214. :
215. :   if (secret_size<=0) {
216. :     return NULL;
217. :   }
218. :
219. :   if (!(bytes = malloc(secret_size))) {
220. :     return NULL;
221. :   }
222. :
223. :   fgets(line, LINE_LENGTH, fp);
224. :
225. :   /* If a secret name is specified, seek to the appropriate label */
226. :   if (secret_label!=NULL) {
227. :     found=FALSE;
228. :     while (!feof(fp) && !ferror(fp) && !found) {
229. :       if (is_label(line)) {
230. :  secretIndex = get_label(line);
231. :  if (secretIndex >=0 && secretIndex == atoi(secret_label)) {
232. :    found=TRUE;
233. :  }
234. :       }
235. :       fgets(line, LINE_LENGTH, fp);
236. :     }
237. :   }
238. :   else {
```

```
239. :     /*  Read empty or comment lines */
240. :     while ((is_empty(line) || is_comment(line)) && !ferror(fp)) {
241. :       fgets(line, LINE_LENGTH, fp);
242. :     }
243. :     if (is_label(line)) {  /* If no secret name was specified,
244. :                               ignore first label */
245. :       fgets(line, LINE_LENGTH, fp);
246. :     }
247. :   }
248. :
249. :   /* Read each line of the hex data until EOF or next label */
250. :   offset = 0;
251. :   while(offset<secret_size && !ferror(fp) && !is_label(line) && !feof(fp))
252. :     {
253. :       /* skip empty lines and comments; translate all other lines */
254. :       if(!is_empty(line) && !is_comment(line))
255. : offset += xlate_line(line, bytes, secret_size, offset);
256. :
257. :       fgets(line, LINE_LENGTH, fp);
258. :     }
259. :
260. :   /* Return the results */
261. :   if(ferror(fp) || offset!=secret_size)
262. :     {
263. :       fclose(fp);
264. :       free(bytes);
265. :       bytes = NULL;
266. :       return NULL;
267. :     }
268. :
269. :   fclose(fp);
270. :   return bytes;
271. : }
272. :
273. :
274. : int get_params(int argc, char *argv[], char **agent_name,
275. :         char **secret_file, char **secret_num,
276. :         char **serviceFile, int *const_debugauth)
277. :      /*  Parses command line parameters.
278. :
279. :  Returns 1 if all required parameters were parsed,
280. :  0 otherwise.
281. :
282. :  Usage: agent_test <auth.agent> -k <secret.txt> [-n <secret_num>]
283. :         -s <service.id>
284. :
285. :  -k: File containing secret key.
286. :  -l:  Label for secret in secret file.
287. :          If not specified, first key is used
288. :  -s: File containing service_id.
289. :  -c: const debug authentication code
290. :     */
291. : {
292. :   int i;
293. :
294. :   *agent_name  = NULL;
295. :   *secret_file = "secrets.txt";
296. :   *secret_num  = 0;
297. :   *serviceFile = "service.id";
298. :   *const_debugauth  = 0;
299. :
300. :   for(i = 1; i < argc; i++)
301. :     {
302. :       if(strcmp(argv[i], "-k") == 0)
303. :  *secret_file = argv[++i];
304. :
305. :       else if(strcmp(argv[i], "-l") == 0)
306. :  *secret_num = argv[++i];
307. :
308. :       else if(strcmp(argv[i], "-s") == 0)
309. :  *serviceFile = argv[++i];
310. :
311. :       else if(strcmp(argv[i], "-c") == 0)
312. :  *const_debugauth = 1;
313. :
314. :       else if(!*agent_name)
315. :  *agent_name = argv[i];
316. :
317. :       else return 0;
318. :     }
319. :
```

```
320. :   /* If i > argc, then an option is missing a parameter that was expected */
321. :   if(i > argc)
322. :     return 0;
323. :
324. :   /* The shell dll file name are required */
325. :   if(!*secret_file)
326. :     return 0;
327. :
328. :   /* The secret file name are required */
329. :   if (!*serviceFile)
330. :     return 0;
331. :
332. :   /* The agent file name are required */
333. :   if (!*agent_name)
334. :     return 0;
335. :
336. :   return 1;
337. : }
338. :
339. : int main(int argc, char *argv[])
340. : {
341. :   DSSPublicKey publicKey;
342. :   char *agent_name=NULL;
343. :   char *secret_file = NULL;
344. :   char *service_file = NULL;
345. :   char *secret_num = NULL;
346. :
347. :   char *key = NULL;
348. :   char *service=NULL;
349. :   char *clientid = NULL;
350. :   char *auth = NULL;
351. :   char session[SESSION_ID_SIZE];
352. :
353. :   int secret_bytes=AGENT_SECRET_LENGTH;
354. :   int i, st, const_debugauth, auth_length, clientid_length;
355. :   int serviceid_length;
356. :   int err;
357. :
358. :   HINSTANCE hDLLAgent = 0;
359. :   AGENT_HANDLE hAgent = 0;
360. :
361. :   AUTHENTICATION_AGENT agentf = NULL;
362. :
363. :   /* Initialize the non-crypto pseudo random number generator. */
364. :   srand(time(NULL));
365. :
366. :   if (!get_params(argc, argv, &agent_name, &secret_file, &secret_num,
367. :         &service_file, &const_debugauth))
368. :     {
369. :       printf("\nusage: agent_test auth.agent [-k secrets.txt]"
370. :       " [-l secret_num]\n\t\t [-s service.id] [-c]\n\n");
371. :
372. :       printf("\t-k: File containing secret key      (default: secrets.txt)\n");
373. :       printf("\t-l: Label for secret in secret file (default: 0)\n");
374. :       printf("\t-s: File containing service_id      (default: service.id)\n");
375. :       printf("\t-c: Get constant debug authentication code\n");
376. :       return 0;
377. :     }
378. :
379. :   /* Pick a session id.  This should be unique for every call to the
380. :      agent: a monotonic counter would work.
381. :
382. :      Here, we use a poor pseudo random number to get 128 bits.  This
383. :      is also sufficient to hinder most attacks. */
384. :
385. :   printf("- Choosing session identifier\n");
386. :   session[0] = '\0';
387. :   if(!const_debugauth)
388. :   {
389. :     for(i = 0; i < SESSION_ID_SIZE-1; i += 2)
390. :       sprintf(session+i, "%2.2x", rand() & 0xff);
391. :   }
392. :
393. :   /* Get the service id.  A server would know this value for all agents. */
394. :   printf("- Loading the agent's service file '%s'\n", service_file);
395. :   service = read_serviceidfile(service_file,&serviceid_length);
396. :   if(!service)
397. :     {
398. :       fprintf(stderr, "%s: unable to read service file\n", argv[0]);
399. :       return 1;
400. :     }
```

```
401. :
402. :   /* Load the agent's secret.  Each agent has its own
403. :      secret. */
404. :   printf("- Loading the agent's secret '%s'\n", secret_file);
405. :   key = read_secretfile(secret_file,secret_num,secret_bytes);
406. :   if(!key)
407. :     {
408. :       fprintf(stderr, "%s: unable to find agent secret\n", argv[0]);
409. :       return 1;
410. :     }
411. :
412. :   /* Get the client id.  A server would retrieve this value for
413. :      registered users. */
414. :   printf("- Calling GetClientID\n");
415. :   clientid_length = GetClientID(GET_CLIENTIDLENGTH);
416. :   if(clientid_length <= 0)
417. :   {
418. :       fprintf(stderr, "%s: unable to get client id length ERR=%d\n",
419. :        argv[0], clientid_length);
420. :       return 1;
421. :   }
422. :
423. :   clientid = malloc(clientid_length);
424. :   if(clientid == NULL)
425. :   {
426. :       fprintf(stderr, "%s: out of memory\n", argv[0]);
427. :       return 1;
428. :   }
429. :
430. :   err = GetClientID(GET_CLIENTID, service, serviceid_length, clientid);
431. :   if(err != 1)
432. :   {
433. :       fprintf(stderr, "%s: unable to get client id ERR=%d\n",
434. :        argv[0], err);
435. :       return 1;
436. :   }
437. :
438. :   /* Load Authentication agent */
439. :   printf("- Loading authentication agent '%s'\n",agent_name);
440. :
441. :   get_public_key(&publicKey);
442. :   st = load_agent(agent_name, &publicKey, &hAgent);
443. :   if(st == TRSAGENT_OK) {
444. :
445. :     /* Since packed agents are marked with a 128-bit universally
446. :        unique ID, there's no real chance that this isn't an agent by
447. :        accident. */
448. :
449. :     /* Get the agent's entry point */
450. :     agentf = (AUTHENTICATION_AGENT) get_agent_proc(hAgent);
451. :     if (agentf == 0) {
452. :       fprintf(stderr,
453. :        "%s: unable to get entry point for %s\n",
454. :        argv[0], agent_name);
455. :       return 1;
456. :     }
457. :   }
458. :   else if(st == TRSAGENT_INVALID_AGENT) {
459. :
460. :     /* Failed to load as a packed agent, try it as a DLL.  */
461. :     hDLLAgent = LoadLibrary(agent_name);
462. :     if (!hDLLAgent) {
463. :       fprintf(stderr, "%s: unable to load authentication agent DLL\n",
464. :        argv[0]);
465. :       return 1;
466. :     }
467. :
468. :     /* Get the agent's entry point. */
469. :     agentf = (AUTHENTICATION_AGENT) GetProcAddress(hDLLAgent, "agent");
470. :     if (!agentf) {
471. :       fprintf(stderr,
472. :        "%s: unable to find entry point in authentication agent DLL\n",
473. :        argv[0]);
474. :       return 1;
475. :     }
476. :   }
477. :   else /* It really did fail badly... */
478. :     {
479. :       char *str = NULL;
480. :       switch(st)
481. : {
```

```
482. :  case TRSAGENT_ALLOC_FAILED:   /* Allocation of memory failed */
483. :    str = "%s: Out of memory\n";
484. :    break;
485. :  case TRSAGENT_DSS_VERIFY_FAILED:/* DSS Signature verification failed */
486. :    str = "%s: DSS Signature verification failed\n";
487. :    break;
488. :  case TRSAGENT_FILE_LOAD_FAILED:
489. :    str = "%s: File load failed\n";
490. :    break;
491. :  case TRSAGENT_DSS_INVALID_KEY:        /* Invalid DSS Key was passed in */
492. :    str = "%s: Attemped to use invalid DSS key\n";
493. :    break;
494. :  default:
495. :    str = "%s: Unknown error code: %d\n";
496. :  }
497. :       fprintf(stderr, str, agent_name, st);
498. :       return 1;
499. :     }
500. :
501. :   /* Call the authentication agent. */
502. :   printf("- Calling authentication agent\n");
503. :
504. :   auth_length = agentf(GET_AUTHLENGTH);
505. :   if(auth_length <= 0)
506. :   {
507. :       fprintf(stderr, "%s: unable to get authentication length\n", argv[0]);
508. :       return 1;
509. :   }
510. :
511. :   auth = malloc(auth_length);
512. :   if(auth == NULL)
513. :   {
514. :       fprintf(stderr, "%s: out of memory\n", argv[0]);
515. :       return 1;
516. :   }
517. :
518. :   if(const_debugauth)
519. :       err = agentf(GET_CONST_DEBUG_AUTHENTICATION, session, auth);
520. :   else
521. :       err = agentf(GET_AUTHENTICATION, session, auth);
522. :
523. :   if(err != 1)
524. :   {
525. :       fprintf(stderr, "%s: authentication agent failed ERR=%d\n",
526. :        argv[0], err);
527. :       return 1;
528. :   }
529. :
530. :   /* Print the authentication code */
531. :   for(i = 0; i < auth_length; i += sizeof(long))
532. :       printf(" %08x", *(long*)(auth+i));
533. :
534. :   printf("\n- Validating authentication code\n");
535. :
536. :   if(ValidateAuthentication(auth, clientid, session, key))
537. :       printf("Validation PASSED\n");
538. :   else
539. :       printf("Validation FAILED\n");
540. :
541. :   free(clientid);
542. :   clientid = NULL;
543. :
544. :   if(auth != NULL)
545. :   {
546. :       free(auth);
547. :       auth = NULL;
548. :   }
549. :
550. :   if(hAgent)
551. :     free_agent(hAgent);
552. :
553. :   if(hDLLAgent)
554. :     FreeLibrary(hDLLAgent);
555. :
556. :   free(service);
557. :   free(key);
558. :   key = NULL;
559. :   return 0;
560. : }
```

Appendix B: References and Further Reading

Books

- Schneier, Bruce. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*
  John Wiley and Sons Publishers, New York
  Second Edition, Copyright 1995

Websites

For information on PGP and GNUPG, see the following websites:

- www.nai.com/default_pgp.asp
- dir.yahoo.com/Computers_and_Internet/Security_and_Encryption (click on PGP)
- www.gnupg.org